JS-作用域篇学习笔记

JavaScript 作用域是一套规则，用于确定在何处以及如何查找变量（标识符）。
这次我们先了解 JS 内部运行的原理：主要分成编译、执行、查询、嵌套、异常五大阶段。

编译阶段

（编译阶段是边解析边执行的）
主要分为分词、解析、代码生成

在传统编译语言的流程中，程序中的一段源代码在执行之前会经历三个步骤，统称为“编译”。

- *分词/词法分析*（Tokenizing/Lexing）
  这个过程会将由字符组成的字符串分解成（对编程语言来说）有意义的代码块，这些代码块被称为词法单元（token）。例如，考虑程序 `var a = 2;`。这段程序通常会被分解成为下面这些词法单元：`var`、`a`、`=`、`2`、`;`。空格是否会被当作词法单元，取决于空格在这门语言中是否具有意义。

分词（tokenizing）和词法分析（Lexing）之间的区别是非常微妙、晦涩的，主要差异在于词法单元的识别是通过有状态还是无状态的方式进行的。简单来说，如果词法单元生成器在判断 a 是一个独立的词法单元还是其他词法单元的一部分时，调用的是有状态的解析规则，那么这个过程就被称为词法分析。

- *解析/语法分析*（Parsing）
  这个过程是将词法单元流（数组）转换成一个由元素逐级嵌套所组成的代表了程序语法结构的树。这个树被称为“抽象语法树”（Abstract Syntax Tree，AST）。
  `var a = 2;` 的抽象语法树中可能会有一个叫作 `VariableDeclaration` 的顶级节点，接下来是一个叫作 `Identifier`（它的值是 a）的子节点，以及一个叫作 `AssignmentExpression` 的子节点。`AssignmentExpression` 节点有一个叫作 `NumericLiteral`（它的值是 2）的子节点。

- *代码生成*
  将 AST 转换为可执行代码的过程被称为代码生成。这个过程与语言、目标平台等息息相关。
  抛开具体细节，简单来说就是有某种方法可以将 `var a = 2;` 的 AST 转化为一组机器指令，用来创建一个叫作 a 的变量（包括分配内存等），并将一个值储存在 a 中。

关于引擎如何管理系统资源超出了我们的讨论范围，因此只需要简单地了解引擎可以根据需要创建并储存变量即可。

“var” ：“keyword”, //关键字
“a” ：“indentifier”, //标识符
“=” ：“assignment”, //分配
“2” ：“intergerr”, //整数
“;” ：“eos”, //(end of statement)结束语句

执行阶段

例如：（var a = 2）
1. 引擎运行代码时首先找当前的作用域，看 a 变量是否在当时的作用域下，如果是，引擎就会直接使用这个变量；如果否，引擎会继续查找该变量；
2. 如果找到了变量，就会将 2 赋值给当前的变量，否则引擎就会抛出异常；

查询阶段

如果查找的目的是对变量进行赋值，那么就会使用 LHS 查询；如果目的是获取变量的值，就会使用 RHS 查询。赋值操作符会导致 LHS 查询。= 操作符或调用函数时传入参数的操作都会导致关联作用域的赋值操作。

JS 引擎首先会在代码执行前对其进行编译，在这个过程中，像 var a = 2 这样的声明会分解成两个独立的步骤：
1. 首先，var a 在其作用域中声明新变量。这会在最开始的阶段，也就是代码执行前进行；
2. 接下来，a =2 会查询（LHS 查询）变量 a 并对其进行赋值。

```
    function foo(a) {
            console.log(a);
     }
     foo(2);
//1.foo()对 foo 函数对象进行 RHS 引用
//2.函数传参 a=2 对 a 进行了 LHS 引用
//3.console.log(a);对 console 对象进行 RHS 引用，并检查其是否有 log()方法
//4.console.log(a);对 a 进行 RHS 引用，并把得到的值传给了 console.log(..
```

嵌套阶段

LHS 和 RHS 查询都会在当前执行作用域中开始，如果有需要（也就是说它们没有找到所需的标识符，）就会向上级作用域继续查找目标标识符，这样每次上升一级作用域（一层楼），最后抵达全局作用域（顶层），无论找到或没找到都将停止。

异常阶段

不成功的 RHS 引用会导致抛出 ReferenceError 异常。不成功的 LHS 引用会导致自动隐式地创建一个全局变量（非严格模式下），该变量使用 LHS 引用的目标作为标识符，或者抛出 ReferenceError 异常（严格模式下）。

```
function foo(a){
                    a = b; // b is not defined
        }
foo(2);

function foo(){
                    var b = 0;
                    b(); // b is not a function
        }
 foo();
```

词法作用域

词法作用域意味着作用域是由书写代码时函数声明的位置来决定的。编译的词法分析阶段基本能够知道全部标识符在哪里以及是如何声明的，从而能够预测在执行过程中如何对它们进行查找。

```
function foo(a) {
        var b = a*2;
      function bar(c) {
                  console.log(a,b,c);
        }
bar(b*3);
}
foo(2);
```

这里面有三层作用域，全局作用域下定义的标识符有 foo；
foo 作用域下定义的标识符有 a、b、bar；
bar 作用域下定义的标识符有 c；
作用域查找从运行时所处的内部作用域开始，逐渐向上进行，直到遇到第一个匹配的标识符为止。
如上面的代码要输出 abc，引擎在 foo 作用域下找到 a 的标识符并赋值为 2；
在 foo 作用域下找到 b 标识符并赋值为 a*2=4；
在 bar 作用域下找到 c 标识符并赋值为 b*3=12；

声明提升

函数时 JS 中最常见的作用域单元。本质上，声明在一个函数内部的变量或函数会在所处的作用域中“隐藏”起来，这是有意为之的良好软件设计原则。所以在多层的嵌套作用域可以定义同名的标识符，这叫做“遮蔽效应”。

我们习惯将 var a = 2; 看作一个声明，而实际上 JS 引擎并不这么认为。它将 var a 和 a = 2 当作两个单独的声明，第一个是编译阶段的任务，而第二个则是执行阶段的任务。

这意味着无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最顶端，这个过程被称为提升。

声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会提升。

声明是的注意事项：
1、 变量的重复声明是无用的，但是函数的重复声明会覆盖前面的声明（后面的函数声明会覆盖前面的函数声明）；
2、 函数声明提升优先级高于变量的声明提升。

总结：要注意避免重复声明，特别是当普通的 var 声明和函数声明混在一起的时候，否则会引起很多危险的问题！

```
function foo(a) {
        var b = a*2;
      function bar(c) {
                console.log(a,b,c);
        }
bar(b*3);
}
foo(2);
```

还是这段代码，全局作用域下声明了 foo 函数；
在 foo 作用域下首先声明了 bar 函数(声明提升)，到 a 变量、到 b 变量；
在 bar 作用域下声明了 c 变量；
在执行输出 abc 时会查找到这些声明后执行赋值操作。

作用域链

自由变量

自由变量：在当前作用域中存在但未在当前作用域中声明的变量；
一旦出现自由变量，就肯定会有作用域链，再根据作用域链查找机制，查找到对应的变量；
查找机制：在当前作用域中发现没有该变量，然后沿着作用域链往上级查找，查到查到对应的变量为止，如果查找不到，直接报错；

```
var a = 1;
var b = 2;
function fn(x) {
        var a = 10;
        function bar(x) {
                var a = 100;
                b = x + a;
                return b;
            }
            bar(20);
            bar(200);
}
fn(0);
```

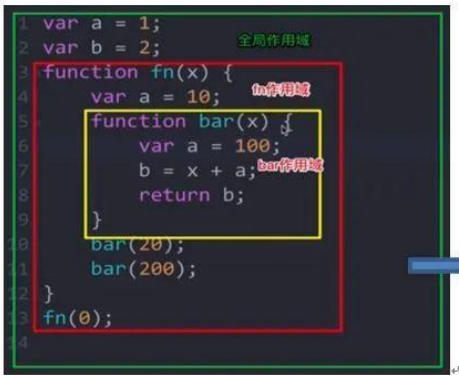
这里面的就存在着一条作用域链 bar => fn => 全局
return b 会根据这条作用链一级一级往上查找到对应的变量；

执行环境

执行流

执行环境也叫执行上下文，执行上下文环境;
每个执行环境都有一个与之关联的变量对象，环境中定义的函数和变量都保存在这个对象;

下面的是对于上面的代码进行执行环境的详细分析：
（arguments 是函数内的固有变量，以数组的形式保存了调用方给该函数传入的所有参数。）



解析

执行

全局执行环境
a:undefined
B:undefined
Fn:function
This:window

第 2 行

全局执行环境
a:1
B:2
Fn:function
This:window

第 13 行

Fn(0)执行环境	全局执行环境
X:0	a:1
a:undefined	B:2
bar:function	Fn:function
Argument:[0]	This:window
This:window	

第 4 行

Fn(0)执行环境	全局执行环境
X:0	a:1
a:10	B:2
bar:function	Fn:function
Argument:[0]	This:window
This:window	

第 10 行

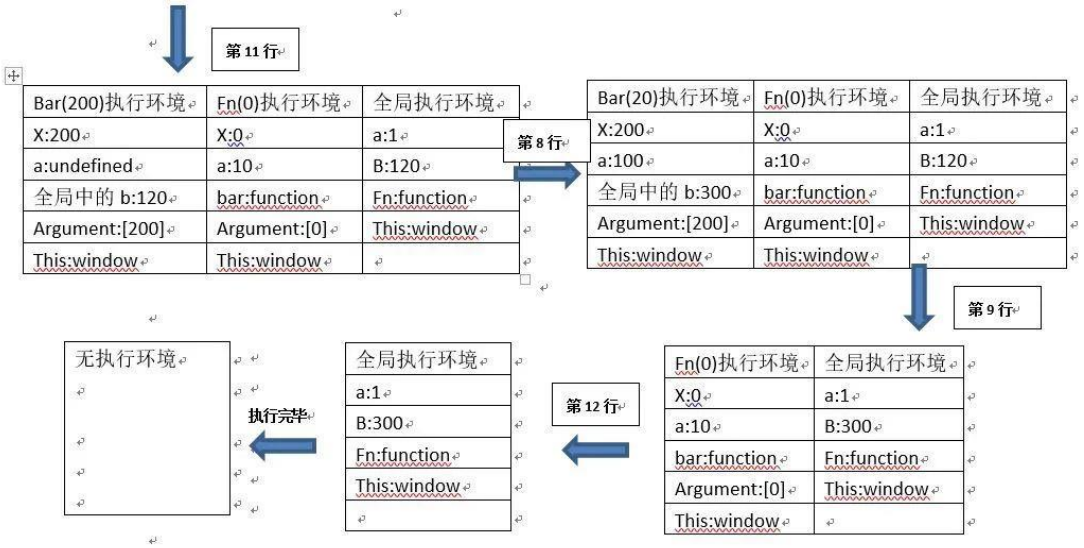
Bar(20)执行环境	Fn(0)执行环境	全局执行环境
X:20	X:0	a:1
a:undefined	a:10	B:2
全局中的 b:2	bar:function	Fn:function
Argument:[20]	Argument:[0]	This:window
This:window	This:window	

第 8 行

Bar(20)执行环境	Fn(0)执行环境	全局执行环境
X:20	X:0	a:1
a:100	a:10	B:2
全局中的 b:120	bar:function	Fn:function
Argument:[20]	Argument:[0]	This:window
This:window	This:window	

第 9 行

Fn(0)执行环境	全局执行环境
X:0	a:1
a:10	B:120
bar:function	Fn:function
Argument:[0]	This:window
This:window	


第 11 行
Bar(200)执行环境 Fn(0)执行环境 全局执行环境
X:200 X:0 a:1
a:undefined a:10 B:120
全局中的 b:120 bar:function Fn:function
Argument:[200] Argument:[0] This:window
This:window This:window
第 8 行
Bar(20)执行环境 Fn(0)执行环境 全局执行环境
X:200 X:0 a:1
a:100 a:10 B:120
全局中的 b:300 bar:function Fn:function
Argument:[200] Argument:[0] This:window
This:window This:window
第 9 行
Fn(0)执行环境 全局执行环境
X:0 a:1
a:10 B:300
bar:function Fn:function
Argument:[0] This:window
This:window
第 12 行
全局执行环境
a:1
B:300
Fn:function
This:window
执行完毕
无执行环境